第8学时　函　　数

几乎所有的计算机语言都支持函数。函数是一组代码，可以按名字对它进行调用，以便执行某项工作，然后返回某个值。在本书中，你要使用许多函数，比如，你已经使用了 print、reverse、sort、open、close和split等函数。它们都是Perl的内置函数。

Perl还允许你编写自己的函数。在 Perl中，用户定义的函数称为子例程。与 Perl的内置函数一样，用户定义的函数也可以拥有参数，并且可以将值返回给调用者。

Perl还支持作用域的概念。作用域用于确定某个时间内程序能够看到的一组变量。由于有了Perl的作用域，你就能够编写运行时不受你的程序的其余部分影响的函数。编写得非常出色的函数可以在其他程序中重复使用。

在本学时中，你将要学习：

- 如何定义你自己的函数和如何调用这些函数。
- 如何将值传递给函数，然后返回值。
- 如何使用use strict来编写程序，以便强制使用某种结构。

8.1　创建和调用子例程

可以使用下面的代码来创建用户定义的子例程：

```
sub subroutine_name {
    statement1;
    :
    statementx;
}
```

Perl中的子例程名与第 2学时中介绍的标量、数组和哈希结构的命名约定是相同的。子例程与现有的变量可以使用相同的名字。但是，你应该避免创建名字与 Perl的内置函数和运算符相同的子例程。如果在 Perl中创建了名字相同的两个子例程，那么在报警特性激活的情况下，Perl就会发出一条警告消息，否则第二个定义的名字会使第一个名字被忘记。

当子例程被调用时，子例程的代码启动运行，并且任何返回值均被重新传递到子例程被调用时的位置。（调用子例程和返回值的内容将在后面介绍。）例如，下面这个短子例程将提示用户输入一个答案：

```
sub yesno {
    print "Are you sure (Y/N)?";
    $answer=<STDIN>
}
```

若要调用一个子例程，可以使用下面两个语句行中的一个：

```
&Yesno();
```

或者

```
Yesno();
```

如果代码中已经声明了子例程，那么可以使用第二个语句（不带 &）；&yesno（）语句是任何位置上都能使用的。在本书中，我们将使用不带 &的语句形式，虽然两种语句形式都可以使用。

当子例程被调用时，Perl能够记住它是在什么位置被调用的，并执行子例程的代码，然后，当子例程运行完成时，返回它记住的程序中的位置，如下面这个例子所示：

```
sub  countdown {
    for($i=10; $i>=0; $i--) {
        print "$i -";
    }
}
print "T-minus: ";
countdown();
print "Blastoff";
```

Perl的子例程可以在程序中的任何位置进行调用，包括在其他子例程中进行调用，如下所示：

```
sub world {
    print "World!";
}
sub hello {
    print "Hello, ";
    world();
}
hello();
```

8.1.1 返回子例程的值

子例程并不只是用于按照一个便于使用的名字将代码组合在一起。子例程与 Perl的函数、运算符和表达式一样，它也有一个值。这个值称为子例程的返回值。子例程的返回值是子例程中计算的最后一个表达式的值，或者是 return语句显式返回的值。子例程的值是在子例程被调用时计算的，然后该返回值将用于调用的任何子例程中。现在请看下面这个代码：

```
sub two_by_four {      # A silly subroutine
    2*4;
}
print 8*two_by_four();
```

在上面这个代码段中，若要使 Perl计算表达式8*two_by_four（）的值，那么子例程 two_by_four（）便开始运行，并返回值8。然后计算表达式8*8，并输出64。

值也可以由子例程的return语句显式返回。当你的程序需要在子例程结束之前返回，或者当你想要明确知道返回的是什么值，而不是“ 堕入 ”子例程的结尾并使用最后的表达式的值时，就需要使用return语句。下面这个代码段同时使用了两种方法：

```
sub x_greaterthan100 {
    # Relies on the value of $x being set elsewhere
    return(1) if ($x>100);
    0;
}
$x=70;
if (x_greaterthan100()) {
    print "$x is greater than 100\n";
}
```

子例程能够返回数组和哈希结构，也能返回标量，如下所示：

```
sub shift_to_uppercase {
    @words=qw( cia fbi un nato unicef );
    foreach(@words) {
       $_=uc($_);
    }
    return(@words);
}
@acronyms=shift_to_uppercase();
```

8.1.2 参数

上面的所有子例程举例都有一个共同点，那就是它们都对硬编码的数据（ 2*4）或者变量进行操作，而这些变量里边恰好拥有正确的数据（ x_greaterthan100（）的$x)。这个限制条件产生了一个问题，因为如果函数依赖硬编码的数据，或者希望得到函数之外的值的数据，那么这样的函数并不是真的能够移植的函数。当你调用函数时说：“ 取出这个数据并且用它进行某些操作 ”，然后在以后又调用它并且说：“ 取出另一些数据并且用它进行某些操作。” 这样，函数的运行特性就可以根据传递给它的值来改变。

为了改变函数的运行特性而赋予函数的这些值称为参数，在本书中你都需要使用这些参数。Perl的内置函数（grep、sort、reverse和print等）都拥有一些参数，并且现在你的函数也可以拥有参数。若要传递子例程的参数，可以使用下面任何一个语句：

```
subname(arg1, arg2, arg3);
subname arg1, arg2, arg3;
&subname(arg1, arg2, arg3);
```

只有当Perl已经遇到子例程的定义时，才能使用上面不带括号的第二种参数形式。

在子例程中，被传递的参数可以通过 Perl的特殊变量@_来访问。下面这个代码段显示了为函数传递参数（3个字符串直接量）和输出参数的情况：

```
sub printargs {
    print join(',', @_);
}
printargs('market', 'home', 'roast beef');
```

若要像下面这个例子中那样，访问传递过来的各个参数，可以使用数组 @_上的下标，就像你对任何其他数组操作时那样。请记住， $_[0]（@_的一个下标）与标量变量$_毫不相干：

```
sub print_third_argument {
    print $_[2];
}
```

对$_[3]这样的变量名进行操作并不是一种 “ 明确的 ” 编程风格。拥有多个参数的函数常常为这些参数赋予一个名字，这样，就能够清楚地知道它们能够做些什么。为理解这些话的含义，请看下面这个例子：

```
sub display_box_score {
    ($hits, $at_bats)=@_;
    print "For $at_bats trips to the plate, ";
    print "he's hitting ", $hits/$at_bats, "\n";
}
display_box_score(50, 210);
```

在上面这个子例程中，数组 @_被拷贝到列表($hits，$at_bats）中。@_的第一个元素$_[0]变成了$hits，第二个元素变成了$at_bats。这里使用的变量名只是为了增强可读性。

变量@_实际上包含了传递给子例程的原始参数的别名。如果修改了 @_（或者修改了@_的任何元素），就会修改参数列表中的元素变量。如果突然进行这样的修改，将被视为一种不好的做法，你的函数不应该干扰来自函数调用者的参数，除非函数的使用者要求这样做。

8.1.3 传递数组和哈希结构

传递给子例程的参数不一定是标量。你可以将数组和哈希结构传递给子例程，但是这样

做需要三思而后行。将数组或哈希结构传递给子例程时使用的方法与传递标量的方法一样：

```
@sorted_items=sort_numerically(@items);
```

在该子例程中，整个数组@items通过@_进行引用：

```
sub sort_numerically {
    print "Sorting...";
    return(sort { $a <=> $b} @_);
}
```

当将数组和哈希结构传递给子例程时，会遇到一个小小的困难。将两个或多个哈希结构（或数组）传递给子例程，通常并不执行你想要做的操作。请看下面这个代码段：

```
sub display_arrays {
    (@a, @b)=@_;
    print "The first array: @a\n";
    print "The second array: @b\n";
}
display_arrays(@first, @second);
```

@first和@second两个数组一道被放入一个列表，各个元素则在调用子例程时放入 @_中。@first的各个元素的结尾与 @_中的@second的元素的开始是无法区分的，它只是一个大型平面列表。在子例程中，赋值语句（ @a，@b）=@_取出@_中的所有元素，并将它们赋予 @a。数组@b没有得到任何元素。（其原因已经在第4学时中做了介绍。）

标量的混合体可以用单个数组或哈希结进行传递，只要标量在参数列表中被首先传递，并且知道有多少个标量。这样，哈希结构或数组就包含了最后一个标量以外的所有值，正如下面这个例子中的情况一样：

```
sub lots_of_args {
    ($first, $second, $third, %hash)=@_;
    # rest of subroutine…
}
lots_of_args($foo, $bar, $baz, %myhash);
```

如果必须将多个数组和哈希结构传递到一个子例程中（并且能够在以后对它们进行区分），则必须使用“ 引用 ”。我们将在第13学时中介绍如何传递引用的方法。

8.2 作用域

在本学时的开头，我们介绍了子例程被用来取出一些代码片，并将它们捆绑在一起，再给它们赋予一个名字。然后就可以使用这个名字在需要的时候执行该代码。子例程还允许你取出子例程中的代码，并使它独自运行。这就是说，你可以让它只使用它的参数、该语言的内置函数和表达式来运行，以产生一个返回值。然后你可以在其他程序中重复使用该函数，因为该函数不再依赖它被调用时所在的上下文，它只是取出它的参数，即内部数据，然后产生一个返回值。该函数将变成一个黑匣子，数据可以进去，也可以出来，你不必从外面关心发生了什么事情。这称为纯函数。

现在请看下面两个代码段：

```
# One fairly good way to write this function
sub moonweight {
    ($weight)=@_;
    return($weight/6);
}
print moonweight(150);
```

```
# A poor way to write this function.
sub moonweight {
    return($weight/6);
}
$weight=150;
print moonweight;
```

从长远来看，上面显示的第一个代码段在某种程度上说要好一些。它不要求设置任何外部变量，即函数外边的变量。它使用它的参数（它拷贝到 $weight的参数），然后进行计算。第二个实现代码不容易在另一个程序中重复使用，必须确保 $weight已经正确地进行设置，并且没有用于某些其他值。如果它被用于其他某个值，你就必须编辑 moonweight（）函数，以便使用一个不同的变量。这样做并不非常有效 。

因此，上面的第一个例子是个比较好的函数，但是它仍然忽略了某些东西。变量 $weight可能与程序中的其他某个位置上的名字为 $weight的变量发生冲突。

Perl允许你在大型程序中为了不同的目的一次又一次地重复使用变量名。按照默认设置，在你的程序的主体中和子例程中， Perl的变量是可视的，这些类型的变量称为全局变量。

你要做的工作是使变量成为函数的专用变量。为此，必须使用 my操作符：

```
sub moonweight {
    my $weight;
    ($weight)=@_;
    return($weight/1.66667);
}
```

在moonweight（）中， $weight现在是个专用变量。程序中的其他函数都不能访问 $weight的值。带有$weight名字的任何其他变量均与 moonweight（）函数的$weight完全隔开。这个子例程现在已经完全是个独立的子例程。

可视变量的这部分程序称为变量的作用域。

可以使用my操作符来声明标量、数组和哈希结构的变量是子例程的专用变量。例如，文件句柄、子例程和 Perl的特殊变量$!、$_和@_都不能标记为子例程的专用变量。如果将括号用于my操作符，你就能够声明多个专用变量：

```
my($larry, @curly, %moe);
```

子例程的专用变量与全局变量的存储方式是完全不同的。全局变量和专用变量可以拥有相同的名字，但是它们互相之间毫不相干，如下所示：

```
sub myfunc {
    my $x;
    $x=20;      # This is a private $x
    print "$x\n";
}
$x=10;          # This is a global $x
print "$x\n";
myfunc();
print "$x\n";
```

上面这个代码段将输出 10、20，然后输出10。myfunc（）子例程中的$x与该子例程外面的$x是完全不同的。子例程有可能同时使用它的专用 $x和全局$x吗？答案是肯定的，不过答案有些复杂，这个问题不在Perl入门书籍要讲解的范围之内。

大部分时候， Perl子例程首先要将 @_赋予一个变量名的列表，然后声明该列表是子例程的专用列表：

```
sub player_stats {
```

```
    my($at_bats, $hits, $walks)=@_;
    # Rest of function…
}
```

这种方法能够创建一个与程序员友好的函数，它的变量都是函数的专用变量，因此它们不会影响其他的函数，或者受其他函数的影响（包括程序的主体）。当子例程运行结束时，所有专用变量均被撤消。

my操作符的其他用法

你为变量声明的作用域也可以小于这个子例程。 My操作符声明的变量的作用域实际上可以包含一个属于子例程块的代码块。例如，在下面这个代码段中，专用变量 $y（用my声明的这个变量）只有在该代码块中才能看到：

```
$y=20;
{
    my $y=500;
    print "The value of \$y is $y\n";    # Will print 500
}
print "$y\n";                            # Will print 20.
```

这个声明甚至可以在控制结构 for、foreach、while或if中出现。实际上，在你拥有代码块的任何地方，都可以为变量设定作用域，这样，它就只能在该代码块中才可以看到，如下面这个例子中那样：

```
while($testval) {
    my $stuff;      # Only visible within the while() loop.
    :
}
foreach(@t) {
    my %hash;       # Only visible within the foreach loop.
}
```

在上面这个代码段中，每次通过这个循环时，便会创建 my的新变量$stuff和%hash。

Perl的5.004和更新的版本允许将 for和foreach循环中的迭代器以及 while和if中的测试条件声明为代码块的专用迭代器和测试条件：

```
foreach my $element (@array) {
# $element is only visible in the foreach()
}

while(my $line=<STDIN>) {
    # $line is visible only in the while()
}
```

同样，当包含的代码块运行结束时，代码块的任何专用变量及其值均被撤消。

8.3 练习：统计数字

既然你已经懂得了函数的基本概念，那么应该开始进一步了解将代码封装在独立函数中的好处。函数提供了可以很容易重复使用的代码。在下面这个练习中，有 3个函数能够对几个数字组进行分析。

让我们回顾一下在学校中学习过的一些知识。一组数字的平均值（也称为算术平均值）只是所有数字的平均值。中项值是你给一组数字排序时位于中间的这个数字的值。如果元素的数量是偶数，那么中项值是位于中间的两个数字的平均值。标准偏差的概念是指平均值附

近的数字“聚合”有多密。大标准偏差意味着数字分布得很宽，而小标准偏差则意味着数字在平均值附近聚合得很紧密。平均值加上或减去标准偏差用于代表大约 68%的数字集，而平均值加上或减去两个标准偏差则代表95%的数字集。

使用文本编辑器，键入程序清单8-1的程序，将它保存为stats。务必根据第1学时中的说明使该程序成为可执行程序。

当完成上述操作后，键入下面的命令，以运行该程序：

```
perl stats
```

程序清单8-1 Stats程序的完整清单

```
1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  sub mean {
5:          my(@data)=@_;
6:          my $sum;
7:          foreach(@data) {
8:                  $sum+=$_;
9:          }
10:         return($sum/@data);
11: }
12: sub median {
13:         my(@data)=sort { $a <=> $b} @_;
14:         if (scalar(@data)%2) {
15:                 return($data[@data/2]);
16:         } else {
17:                 my($upper, $lower);
18:                 $lower=$data[@data/2];
19:                 $upper=$data[@data/2 - 1];
20:                 return(mean($lower, $upper));
21:         }
22: }
23: sub std_dev {
24:         my(@data)=@_;
25:         my($sq_dev_sum, $avg)=(0,0);
26:
27:         $avg=mean(@data);
28:         foreach my $elem (@data) {
29:                 $sq_dev_sum+=($avg-$elem)**2;
30:         }
31:         return(sqrt($sq_dev_sum/@data-1));
32: }
33: my($data, @dataset);
34: print "Please enter data, separated by commas: ";
35: $data=<STDIN>;  chomp $data;
36: @dataset=split(/[\s,]+/, $data);
37:
38: print "Median: ", median(@dataset), "\n";
39: print "Mean: ", mean(@dataset), "\n";
40: print "Standard Dev.: ", std_dev(@dataset), "\n";
```

第1行：这一行包含了到达解释程序的路径（可以改变这个路径，使之适合系统的需要）和开关-w。请使警告特性始终处于激活状态。

第3行：命令use strict意味着所有变量必须用my来声明，裸单词必须用引号括起来。

第4~11行：使用foreach循环，使mean（）函数运行，将$sum中的所有数字相加，然后除

以数字的数量。

第12~21行：median（）函数以两种方式运行。如果元素是个奇数，它只是选择中间的元素，方法是取出数组的长度，再除以 2，然后使用它的中间部分。如果元素的数量是偶数，它进行相同的操作，但是它取出两个中间的数字，然后使用 mean（）函数计算这两个在 $upper和$lower中的数字的平均数，并将它作为中间值返回。

第23~32行：std_dev（）函数非常简单，不过它主要是个数学运算函数。简单说来，从平均数中减去@data中的每个元素值，再求它的平方值，然后将产生的结果在 $sq_dev_sum中累加。若要求得标准偏差，则用平方差的合计值除以元素的数量减 1，然后求它的平方根。

第33~35行：程序的主体中需要的变量被声明为词法单位（使用 my进行声明），并提示用户输入$data。然后使用模式/[\s,]+/将变量$data分割成数组@dataset。该模式按照逗号和空格对该行进行分割。额外的空格和逗号被忽略。

第38~40行：产生输出。请注意，这并不是调用函数 mean（）、median（）和std_dev（）的惟一位置。这些函数也可以互相调用， std_dev（）与median（）同时使用mean（），这是重复使用代码的很好的例子。

程序清单8-2给出了一个Stats程序输出的示例。

程序清单8-2 Stats程序的输出示例

```
Please enter data, separated by commas: 14.5,6,8,9,10,34
Median: 9.5
Mean: 13.5833333333333
Standard Dev.: 10.3943093405318
```

8.4 函数的脚注

现在你已经懂得了作用域的概念，有些操作只能用作用域才能有效地进行。可用函数之一是递归函数，另一个是 Perl语句use strict，它能够激活更严格的 Perl，使你能够避免在编程中出现错误。

8.4.1 声明local变量

Perl 的第4版并不配有真正意义上的“专用”变量。相反， Perl 4的变量只能称为“准”专用变量。这个“准”专用变量的概念在 Perl 5中仍然存在。若要声明这些变量，可以像下面这个例子中那样，使用local操作符：

```
sub myfunc {
    local($foo)=56;
    # rest of function…
}
```

在上面这个代码段中， $foo被声明为myfunc（）子例程的局部变量。用 local声明的变量的作用与使用my声明的变量几乎相同，它的作用域可以局限于一个子例程，一个代码块，或者eval，它的值将在退出子例程或代码块时被撤消。

它们之间的差别是：声明为局部变量的那些变量，可以在它的作用域范围内的代码块中看到，也可以在从该代码块中调用的任何子例程中看到。表8-1显示了这两种变量之间的逐项比较。

表中显示的两个代码段基本相同，差别在于 myfunc（）中对$foo的声明不一样。在左边，它用my进行声明，而在右边，它用 local来声明。

表8-1 my变量与local变量的比较

```
sub mess_with_foo {
    $foo=0;
}
sub myfunc {
    my $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

```
sub mess_with_foo {
    $foo=0;
}
sub myfunc {
    local $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

当左边的代码运行时，创建的$foo是myfunc（）的专用变量。当mess_with_foo（）被调用时，在mess_with_foo（）中被修改的$foo是全局变量$foo。当控制返回给myfunc（）时，输出的值是20，因为myfunc（）中的$foo从来没有改变。

当右边的代码运行时，$foo被创建并声明为myfunc（）的局部变量。当mess_with_foo（）被调用时，$foo被设置为0。该$foo与myfunc（）返回的$foo相同，“专用性”传递到了被调用的子例程。当返回到myfunc（）时，输出0这个值。

如果你对术语非常讲究，局部变量在技术上可以称为动态作用域变量，因为它们的作用域可以随着被调用的子例程而变化。用 my声明的变量可以称为词法作用域变量，因为它们的作用域只需要通过读取代码并指明它们声明时所在的代码块来决定，该作用域是不变的。

每当你的程序需要子例程专用的变量时，你几乎总是想要一个用 my声明的变量。

8.4.2 使Perl变得更加严格

Perl是一种比较随意的编程语言。它并不试图限制你的编程操作，它允许你在完成工作时不会过多地抱怨代码的外在形式。你也可以使 Perl对你的代码更加严格一些。例如，如果你在命令行或者#!行上使用警告开关，那么 Perl能够帮助你避免犯一些愚蠢的错误。当你使用未定义的变量，或者仅仅使用一次变量名时， Perl就会向你发出警告。

在较大的软件项目中，或者在你的程序变得越来越大的时候，就需要让 Perl帮助你对程序有所约束。除了使用-w开关外，也可以在编译时告诉Perl解释程序打开更多的警告消息。可以使用use strict来进行这项操作：

```
strict:

use strict;
sub mysub {
    my $x;
    :
}
mysub();
```

use strict语句实际上可以称为编译器命令。它能够告诉 Perl，给下列情况做上指向当前代码块或文件中的运行期错误的标志：

- 试图使用不是用my声明的变量名（不是特殊变量的名字）。
- 当函数定义尚未设置时，试图将裸单词用作函数名。
- 其他潜在的错误。

现在，use strict命令能够帮助你避免产生前面两个问题。让 Perl给没有使用my声明的变量做上标志，就可以在你实际上打算使用专用变量时，避免使用全局变量。这是帮助你编写更具独立性的代码并且不依赖全局变量的一种方法。

use strict解决的最后一个问题是裸关键字的问题。请看下面这个代码：

```
$var=value;
```

在这个例子中，你打算将 value解释为一个函数调用还是一个字符串呢（但是你忘记了引号）？Perl的use strict命令会指出这个代码是个含糊的代码，并且不允许使用这种句法，除非在到达该语句之前已经对子例程的值进行了声明。

从现在起，将把use strict命令纳入本书中的所有的练习和较长的程序清单中。

8.4.3 递归函数

你早晚都会遇到一个特殊的子例程类别。这些子例程实际上通过调用它们自己来执行它们的操作。这些子例程称为递归子例程。

每当一些任务能够分割成较小的任务和较小的相同任务的时候，就可以使用递归子例程。比如，一个递归子例程正在搜索一个目录树，以便寻找一个文件。当搜索了最上面的目录后，找到了子目录，因此又必须搜索这些子目录。在这些子目录中，如果又找到了下一个层子目录，那么又必须搜索下一个层子目录。在这里我们可以看到一种模式。

另一种递归任务是计算阶乘，它常常用于统计学。字母 ABCDEF排列方法的数量是6的阶乘。阶乘是一个数与所有较小的数（最小为 1）的乘积。因此，6的阶乘是6×5×4×3×2×1，即720。若要计算6的阶乘，你必须计算5的阶乘，再将它乘以6。若要计算5的阶乘，你必须计算4的阶乘，再将它乘以5，等等。程序清单8-3显示了计算阶乘时使用的递归函数。

程序清单8-3 计算阶乘的递归函数

```
1:   sub factorial {
2:      my ($num)=@_;
3:      return(1) if ($num <= 1);
4:      return($num*factorial($num-1));
5:   }
6:   print factorial(6);
```

第2行：factorial（）子例程的参数被拷贝到$num，它声明为该子例程的专用变量。

第3行：每个递归函数都需要有一个终止条件。也就是说，需要设定一个位置，在这个位置上，该函数不再能够调用自己以便获得一个答案。对于 factorial（）子例程来说，终止条件是1的阶乘（或0的阶乘）。这两个阶乘的值都是 1。当使用1或0（$num<=1）调用factorial（）子例程时，它就执行return（1）。

第4行：否则，如果参数不是 0或1，那么必须计算下一个较小的数列的阶乘。如果 $num分别是：6、5、4、3、2、1，则第4行就分别计算下面的值：返回（6×阶乘（5））、返回（5×阶乘（4））、返回（4×阶乘（3））、返回（3×阶乘（2））、返回（2×阶乘（1））、不能到达第4行。factorial（1）返回1。

当下一个较小的阶乘计算后（一直计算到 1），该函数开始返回上面各个函数调用序列的值，直到最后能够计算6的阶乘为止。

递归函数并不是个常用的函数。大的递归函数创建起来非常复杂，而且很难调试。凡是可以通过迭代（使用 for、while和 foreach）来执行的任务都可以使用递归函数来进行操作，同时，任何递归任务也可以通过迭代来完成。递归函数通常保留用于执行少数任务，目的是使它们执行起来容易一些。

8.5 课时小结

Perl支持用户定义的函数，这种函数称为子例程，子例程的运行特性与内置函数相同，它们可以带有参数，执行操作，然后在必要时将各个值返回给调用者。 Perl的函数能够调用其他函数，甚至能够调用它们自己。Perl还允许声明对一个函数（或任何代码块）来说专用的变量，并能创建可以重复使用的独立的代码块。

8.6 课外作业

8.6.1 专家答疑

问题：在调用函数时，使用和不使用 &有没有真正的区别？

解答：现在你还没有必要考虑这个区别。当使用函数的原型或者当你调用的函数没有括号时，在$foo与foo之间存在一个很小的差别。这些问题不在本书讲解的范围之内，但是为了满足用户的好奇心，在perlsub手册页中，对这个问题做了介绍。

问题：当我在程序中使用my（$var）时，Perl报了一条消息：syntax error，next 2 tokens my(，这是什么意思？

解答：可能出现了键入错误，也可能安装了 Perl 4这个版本。请在命令提示符后面键入perl -v。如果Perl报出的版本是第4版，那么你应该立即进行版本升级。

问题：应该如何将函数、文件句柄、多个数组或哈希结构传递（或返回）给子例程？

解答：若要传递函数、多个数组和哈希结构，必须使用引用，这个问题将在第 13学时中介绍。若要将文件句柄传递给子例程，或者从子例程那里接收文件句柄，必须使用称为typeglob的工具，或者使用IO::Handle模块。这两个内容都不属于本书讲解的范围。

问题：我使用的一个函数返回了许多值，但是我只对其中的一个值有兴趣，我应该如何跳过其他的返回值？

解答：方法之一是用函数建立一个列表，方法是将整个函数调用放在括号中。当它是个列表时，你就可以使用普通的列表块来获得该列表的各个部分的信息。 下面这个代码只是从内置函数localtime（它实际上有9个返回值）中取出年份（当前年份减去 1900）：

```
print "It is now ",1900+ (localtime [5] )
```

另一种方法是将来自函数的返回值赋予一个列表，并且将不需要的值赋予 undef或哑变量：

```
(undef, undef, undef, undef, undef, $year_offset )= localtime;
```

8.6.2 思考题

观察下面这个代码段：

```
sub bar {
    ($a,$b)=@_;
    $b=100;
    $a=$a+1;
}
sub foo {
    my($a)=67;
    local($b)=@_;
    bar($a, $b);
}
foo(5,10)
```

1) 当你运行bar（$a,$b）后，$b中的值是什么？

a. 5

b. 100

c. 68

2) foo（）的返回值是什么?

a. 67

b. 68

c. undef

3) 在foo（）中，$b是什么？

a. 词法规定的作用域

b. 动态作用域

c. 全局作用域

8.6.3 解答

1) 答案是b。$b在foo（）中声明为local，因此每次调用的子例程均共享同一个$b值（除非它们后来再次用local或my声明了$b）。调用bar（）后，在$b被修改的地方，$b被设置为100。

2) 答案是b。foo（）中的最后一个语句是 bar（$a，$b）。bar（）返回68。因为$a的值被传递给bar，并且它被递增了。foo（）返回最后一个表达式的值，它是 68。

3) 答案是b。用local声明的变量是动态调用的作用域变量。

8.6.4 实习

- 使用本学时的统计练习中的函数和第 7学时中的单词统计代码来观察文档中单词的长度。计算它们的平均值、中间值和标准偏差。
- 编写一个函数，输出斐波纳奇数列的一部分。这个数列的开始部分是 0，1，1，2，3，5，8，它可以永远延续下去。斐波纳奇数列是数学和自然界中的一种递归模式。后面这个数是前两个数的和（0和1是例外）。这些数可以用迭代方式和递归方式进行计算。